网易基于Apache Ranger的数据安全中心实践

安全运营 (https://www.secrss.com/articles?tag=安全运营) · DataFunTalk (https://www.secrss.com/articles?author=DataFunTalk) · 2023-06-13

本次分享主题为网易基于Apache Ranger构建大数据安全中心的实践。

导读 本次分享主题为网易基于Apache Ranger构建大数据安全中心的实践。主要内容包括：

主要内容包括以下几大部分：

1. Apache Ranger介绍

2. 大数据安全中心整体解决方案

3. 关键技术分析

4. 成果&规划

分享嘉宾｜吴侯 网易数帆 资深开发工程师

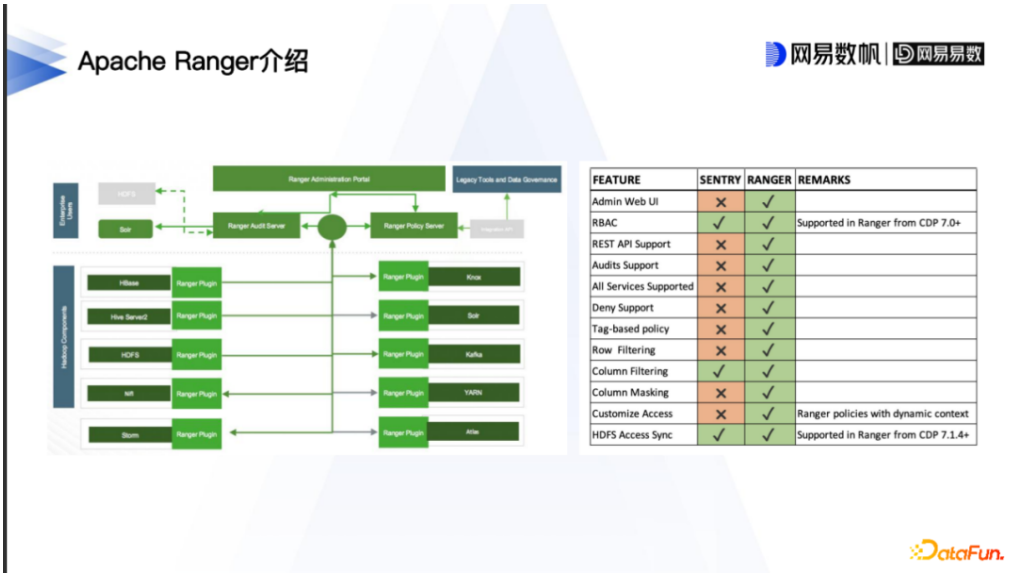
编辑整理｜唐洪超 hotata

出品社区｜DataFun

01 Apache Ranger介绍

1. 总体介绍

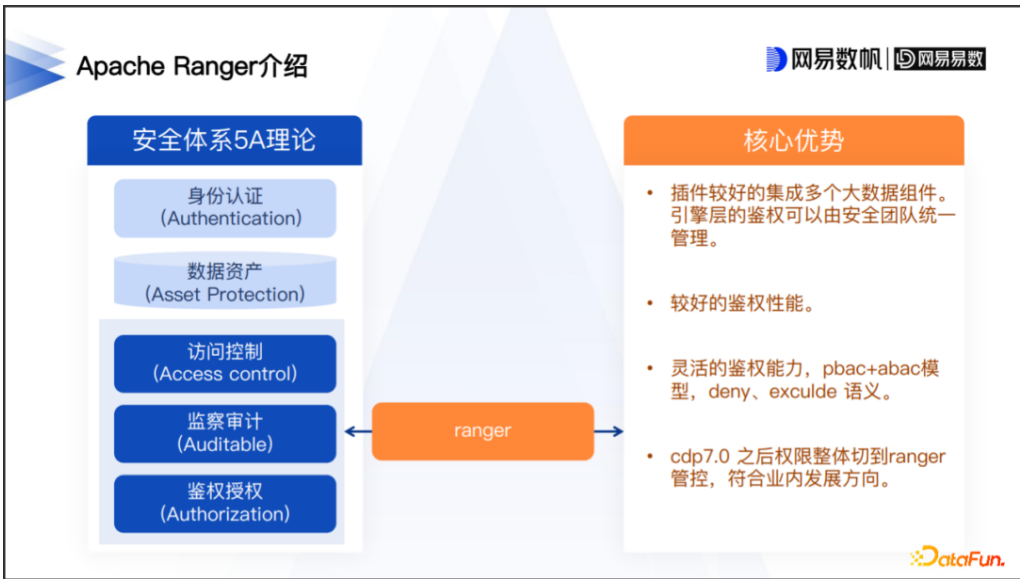
Apache Ranger是Hadoop生态中的一个安全解决方案，它通过在各个组件中集成Ranger 的插件，用中心化的 Ranger Admin来控制所有的policy，插件通过同步policy到本地缓存来提供鉴权的能力。



FEATURE	SENTRY	RANGER	REMARKS
Admin Web UI	✗	✓	
RBAC	✓	✓	Supported in Ranger from CDP 7.0+
REST API Support	✗	✓	
Audits Support	✗	✓	
All Services Supported	✗	✓	
Deny Support	✗	✓	
Tag-based policy	✗	✓	
Row Filtering	✗	✓	
Column Filtering	✓	✓	
Column Masking	✗	✓	
Customize Access	✗	✓	Ranger policies with dynamic context
HDFS Access Sync	✓	✓	Supported in Ranger from CDP 7.1.4+

上方右图是Ranger 的能力列表。可以看到 Ranger 的各种能力，包括管理行级权限，tag base policy，列级权限，列级过滤，列级脱敏等等能力。

接下来看一下Apache Ranger的优势在哪里。



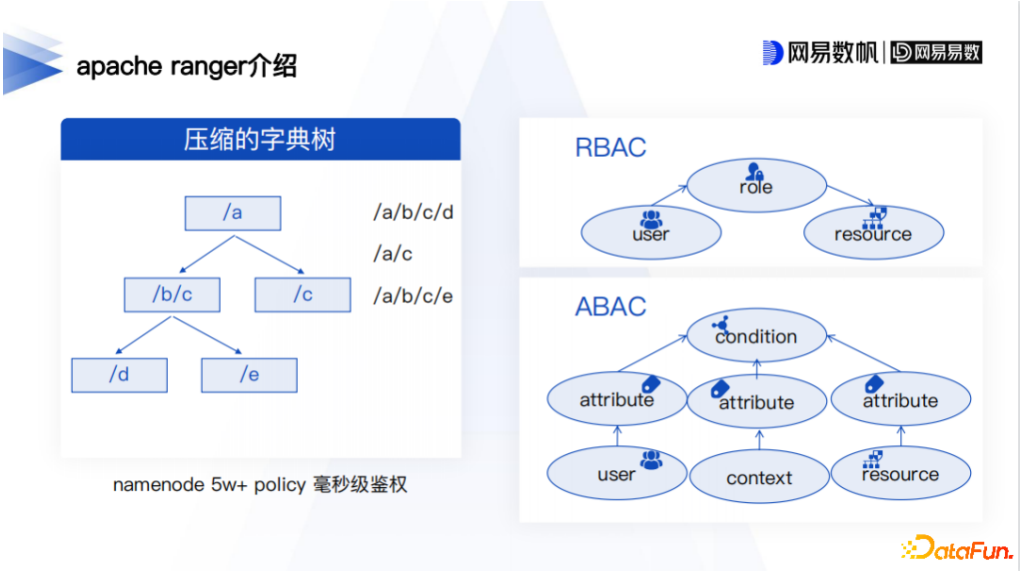
安全体系的5A 理论中， Ranger可以解决其中访问控制、监察审计以及鉴权授权这三块，解决起来的核心优势有四点：

第一个优势是它集成了多个大数据组件鉴权插件，比如Hive、Spark、Presto 等等一大批主流引擎的鉴权的插件。插件对组件做了非常多的适配。如果使用Ranger，可以把所有引擎的鉴权收拢由安全团队统一负责管理。如果不用Ranger，大概率是每个团队独自管理各个组件鉴权逻辑，这样比较分散，也会增加团队之间沟通的成本。

第二个优势是它有较好的鉴权性能，Ranger针对policy缓存构建了大量的索引，可以很好到应对 NN 这种实时性要求高的服务。对于NN来说鉴权几毫秒和几百毫秒会有很大差距。

第三个优势是它有非常灵活的鉴权能力，除了 RBAC以外，还提供 ABAC 的能力。同时还有 deny、exclude语义，鉴权非常灵活，支持的权限模型也比较多。

第四个优势也是最关键的一点，是我们调研发现，业内的 CDP 7.0 之后，整体的权限方案都切换为Ranger，包括业内的一些存储引擎，比如alluxio等等也都在集成Ranger。因此我们有了一个基本的判断，就是在大数据的生态， Ranger已经基本成为一个事实上的标准或者是一个业内主流的发展方向了。为了以后社区安全的新功能更好的集成，我们需要去紧跟Ranger。



左图可以看到Ranger对所有 policy 的资源都做了一个压缩的字典树的索引，极大地提高了针对policy的查询效率。在我们内部的实践中，在5~6万 policy 的情况下，namenode 鉴权依然可以达到毫秒级，性能非常高。

另外Ranger 同时支持 RBAC 和ABAC 两种模型。

这里简单介绍一下两种模型：

RBAC 模型是一个最常用的通过角色来管理资源的权限管理模型，这种模型比较简单，也比较常用。ABAC模型是用户资源以及上下文都会带有属性，最终的权限根据属性计算得来，这种模式比较灵活。深入观察可以发现 RBAC 其实是 ABAC的一种特殊情况，因为可以把角色当成一种特殊的属性。ABAC的优势在于可以让权限做得更精准，比如可以允许所有网易员工早上9点到10点之间访问某个资源，或者是一个IP 段在某个时间点可以访问某个资源。这都是Ranger所支持的一些能力。所以我们可以看到Ranger的能力还是比较强大的。

2. 整体评估

前文介绍了Ranger的优势，不过仅用Ranger是无法完全解决企业中的大数据安全的问题。一个好的安全平台需要做到五点。

- 第一点是安全性，我们需要的是既防君子、也防小人的安全平台，它不同于一般的业务平台主要考虑易用性，还要考虑被恶意攻击或者绕开的问题。
- 第二点是精准性，也就是最小权限原则，能不能给一个用户最小权限，这个权限既不放大也不缩小。权限放大指的是，比如本来不需要给用户某个权限，但是系统架构导致不得不给他额外的授权才能去访问权限。权限缩小一般都是鉴权逻辑导致的问题，比如本来只需要 a 资源权限，但是鉴权逻辑或者权限体系有问题，导致还鉴了另外一个资源的权限，这个时候就得 a 和 b 一起申请才能用。权限系统设计越精准，权限管控越有效。
- 第三点是管理性，是否有利于管理员对整体的企业内部权限有全方位的了解。比如想知道一个资源有哪些人能访问，或者是想知道一个用户能访问什么资源，或者想知道权限快到期的资源有哪些等等。能够支持检索的维度越多，管理性越强，管理员的权限管控体验越好。
- 第四点是效率，比如管理员授权操作链路是否复杂，有大量用户需要授权时是否高效，是否需要专人管理赋权，普通用户能否快速获取到需要的权限等。效率过低，会大大增加管理员的成本，且普通用户也会感觉这套体系很难用，从而拖慢开发的节奏，最终不得不通过牺牲精准性来提高效率。
- 第五点是性能，即鉴权要占用多少资源，要耗费多少时间等等。增加权限后对于整个系统的性能一定是有所下降的，关键看这种影响的程度。

我们来一一对照，看看 Ranger 在这五点上面的表现如何。

第一点安全性。整体而言Ranger 的安全性是比较好的，因为 Ranger 是一个内存鉴权的机制，同时 Ranger 的服务端集成了SSL认证。假设操作系统、Kerberos认证、网络都是安全的，那么我们认为Ranger是很难被绕过的，即使绕过，成本也是非常高的。当然原生Ranger对Spark权限的管理是比较弱的，后面会详细分析。

第二点是精准性。Ranger 的权限生命周期落在 policy 上面，每条 policy 可能会包含非常多的对象，不同的对象可能生命周期又是不同的。使用Ranger这套体系时policy 的生命周期只能按照下面所有授权对象中最大的生命周期来算，否则就会有权限缩小的问题。这种情况导致精准性有所欠缺。另外ranger对于HDFS 删除权限包含在写权限中，没法只授权写而不授权删除。而删除比一般的写风险更大所以删除权限应该要比写权限或者创建权更高，因此可以认为ranger在这点上的精准性也有缺陷。

第三点是可管理性。这一点上Ranger的表现是比较差的。Ranger 的一个特点是，只能根据资源去搜policy，在 policy 中可以搜出它的访问控制列表，但是用其它维度搜索很困难。如果要搜索一个用户所拥有的权限，就必须先把相关所有的 policy 搜出来，再从 policy 中解析出需要的信息。在policy 数量非常大的时候，这种分页查询会受到诸多的限制。再如脱敏的时候，Ranger 的脱敏规则是，policy根据优先级排列，排在前面的优先，排在后面的优先级依次降低。我们很难推断一个用户对某个资源的脱敏规则，因为角色非常多，很难去分析脱敏的整体情况。

第四点效率性。Ranger 也是相对较差的，因为Ranger的体系是依赖于管理员来进行操作的，普通用户是没有任何入口的。Ranger的体系整体上是一个管理员视角的工具，如果用户需要一个权限，只能通过管理员登录Ranger搜索并修改policy。而Ranger 页面搜索性能很低，搜索结果还要再编辑，整体授权链路是很长的。如果一天有很多用户需要申请权限，会非常低效，尤其是团队扩大的时候，低效更加明显。

第五点性能。 前面分析过Ranger 的鉴权性能非常好，但 Ranger policy 对内存占用是较大的，因为Ranger 是全量的缓存policy，内存占用大的情况不仅仅是绝对的数值比较大，由于 Ranger 的policy 的生命周期比较短的，10 秒、15 秒到 30 秒会全量更新一次，会形成非常多短生命周期的内存。比如1G 的内存虽然不算太大，但是1G内存每 10 秒钟都会被全量更新一次，这种情况就会对内存的压力以及 JVM调优产生较大的影响（ranger 2.x之后提供了增量更新，但是还不太成熟容易漏policy）。

从以上分析可以看到 Ranger 的优势以及不足。众所周知，安全性、精准性与管理性、效率、性能必然是矛盾的，如果做得很安全，很精准，那么往往会导致管理性、效率和性能的下降。而管理性、效率、性能很优秀的时候，又必然对安全性和精准性有一定的损失。我们要做的是把这五方面统筹起来，寻找到一个平衡点。

02 大数据安全中心整体解决方案

1. 平台历程

前文中介绍了 Ranger 的优缺点，接下来将介绍网易内部是如何解决这些问题的。首先整体介绍一下网易内部大数据平台安全的发展历程，它主要经历了三个阶段，具体如下图。

（1）阶段一

第一个阶段是 2009 年到 2014 年的一个初始阶段。在这一阶段是采用 Ranger 加 kerberos 加 ldap 去管理权限。Ranger 直接当产品使用，就在Ranger页面上去管理Hadoop集群的权限，其整体就是一个Hadoop的权限工具。

（2）阶段二

第二个阶段是 2014 年到 2021 年，这个阶段的安全管理是数据中台的一个权限模块。这个阶段数据中台对接了Ranger，进行了一些角色用户到中台资源的映射，也构建了一些资源多租户隔离的权限管理方案。这个阶段 Ranger 基本上跟数据中台有了一定的联动，效率得到了一定的提升，但还是存在一些问题。

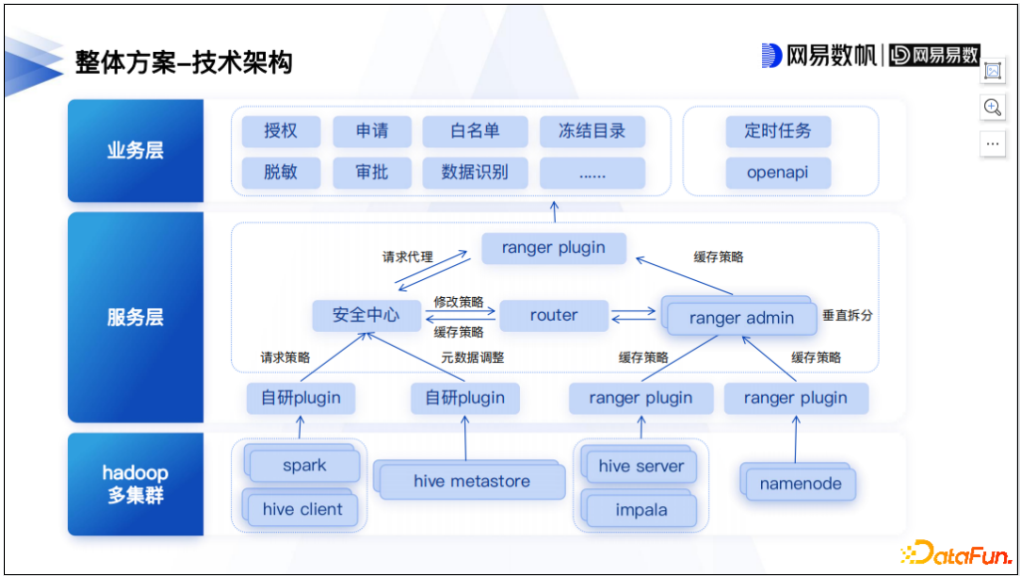
（3）阶段三

第三个阶段是 2021 年到现在，我们构建了一个独立的一站式权限管理平台。它不仅仅是权限管理平台，更是一个安全管理平台，拥有权限脱敏、审计监控、数据扫描识别等多种功能，包括以后的加解密等等。同时我们还做了一个大动作，就是把内部的Ranger从 0. 5 版本升级到 2. 1 版本。

2. 整体方案

下图是整体的产品方案图。

可以看到我们的产品基本上覆盖了安全生命周期中的所有流程，还有一些权限申请、操作审计、行为识别、权限转移、权限释放、数据脱敏、数据加密、白名单等等功能。最后面我们对数据和操作进行了分类分级。下图是一个整体的技术架构，这也是本文要重点分享的。



我们的技术架构基本上分为三层，最下面一层为Hadoop集群，采用多Hadoop集群的模式，各个集群之间网络都可能是隔离的。中间是服务层，数据平台就构建在这一层，主要包括安全中心、Ranger和一些插件。最上面一层是业务层，包括了刚才产品中提到的授权审批、白名单、冻结目录以及一些定时任务等功能。

这套体系中比较重要的一个点就是对客户端的鉴权，比如Spark和Hive的鉴权，采用了自研的plugin。plugin把内存的鉴权转成了网络请求，这样就不用再去缓存过多的内存。同时增加了hive metastore 插件，去根据ddl同步进行Ranger中的权限变更。

对于 Hive server， Impala 以及 namenode，我们基本上采用的是原生的方案，因为对于一个服务来说，内存稍微大并不会产生太大影响。但是Ranger的插件也是经过一些优化的。中间这一层首先对 Ranger admin 进行了一个垂直拆分，比如一个Ranger可能管理若干个集群，且有多个Ranger，这种模式安全中心会将集群路由到指定的Ranger，同时安全中心也会缓存一份policy来做一些鉴权的操作。对于顶层，是一些业务逻辑和提供的能力。

这套架构有四个核心的思路。

（1）低耦合

首先，对 Ranger 的开发，一定要是低耦合的开发。因为过度耦合在软件工程层面会削弱变更的稳定性。我们不依赖Ranger的policy格式和弱字段做业务逻辑。只能依赖 policy 的语义，只要Policy的语义是一样的，上层的逻辑就必须是一样的。比如Ranger的一个 policy 有三个 item另外一个policy有两个item，但是两条policy表达的含义是一样的，此时上层的逻辑也必须是一样的。另外policy中有些弱字段，比如policy描述字段description，这种描述字段不能用它去做强逻辑，只能做展示逻辑。这是我们总结出来的一个比较重要的经验。

这么做的目的首先是不搞约定式的逻辑能够让代码在多个人之间更好维护。同时Ranger admin也可以独立使用，以达到低耦合。否则如果过度耦合，Ranger admin 就相当于变成了一个类似于数据库的存储服务，也就无法对用户暴露出来，只能用我们的平台去管理权限。在我们的实践中，尤其是ToB商业化的过程中，这种情况是不现实的，有些用户对ranger理解可能比较深入，就不期望这样。

（2）低侵入

第二个就是低侵入，就是尽量不去修改Ranger 本身的代码，原因之一是 Ranger 服务端代码是比较老旧的， Ranger采用的还是jpa的一些数据库框架，包括Spring版本都是很老的，这些老旧框架对研发效率的折损是较为严重的，我们现在的微服务等后端技术框架日新月异，如果还用这种老的框架，会导致整体的研发效率大打折扣。其二，如果我们把大量业务逻辑注入在Ranger中，会导致Ranger很难升级，尤其是大版本的升级,比如刚才谈到的，从0. 5版本升级到2. 1版本，成本将非常高。

（3）流量分摊

第三个是一定要做好流量的分摊，不能把所有的流量都打到 Ranger admin 上面，因为Ranger admin 的接口内部有一些缓存更新机制，这些机制都是会加锁的，在流量增大的时候，锁冲突就会非常严重，这时Ranger admin的性能降低，会导致上层业务超时等问题影响客户使用，所以我们要遵循一个核心思路，读的请求尽量不要直接打到Ranger admin上面，尽量做一些额外的缓存策略或者是一些索引策略。Ranger本身其实可查询的维度也不够丰富，如果把所有数据都放到内存里面再去慢慢查，内存消耗将过大。

（4）一致性保证

最后一点就是要保证一致性，比如中台和Ranger之间先调用了中台去存储一个权限，然后再去将权限存储到Ranger的时候失败了，这样就可能导致中台与Ranger的数据不一致，体现到用户视角就是在中台中看到有权限， Ranger中却没有权限。因此一致性是一定要考虑到的，不能只考虑中台操作和ranger同时成功的情况，还要考虑部分成功，部分失败的情况，这样才可以保证系统可用性，提高用户体验。

03 关键技术解析

接下来将介绍10个比较有特色的设计细节。

1. 统一权限检索问题

上文讨论的Ranger的资源，其原生的权限模型有很多优势，但是也存在一些劣势。比如我想知道一个人具备哪些权限，一个脱敏算法到底被哪些地方用到了，这些都是没法搜索的，用户粒度的权限生命周期也没法设置。针对这个问题，网易内部搭建了一层权限检索模型，如下图。

权限检索模型是一个纯粹只读的视图，基于Ranger的 change log进行一个变更的同步，比如 30 秒进行一个变更的同步，同步过来后，会把policy 解析成我们需要的搜索维度和格式存放到在我们的权限模型中，所有的查询接口都通过我们的权限模型，Ranger admin只做写的入口。比如业务层要授权，要申请审批，之后权限调整，修改脱敏规则等等这些修改的操作都只会操作Ranger admin，保证是单写的，不会去先在中台写一份，Ranger再写一份。从而避免上文说的双写不一致的问题。通过这种异步的同步机制，虽然查询可能会有 30 秒的延迟，但是可以保证最终的一致性。

Ranger change log 是2.x版本才推出的一个特性，它记录了所有的policy增删改查的事件，这些变更日志存在一个数据库里面，通过一个接口去拉取。通过这套方案既能保证最终一致性，又能在此基础上极大地提高查询性能，而灵活的查询又使得产品体验提升。各个维度的权限数据用户都可以去检索，前面举例提到的，一个人有哪些权限、哪些权限要到期了等等问题，都可以便捷搜索到结果，管理员对权限整体上有很全面的了解，体验非常好。同时权限生命周期也使用这套权限模型来实现，可以达到精细化的设置权限生命周期的目的。

2. 鉴权优化&多集群

我们在实践中发现1万条 policy 基本上占用内存在 100- 200 M之间，如果有几万条，甚至上 10 万条，量就非常大。再加上，Ranger admin缓存了一份全量的policy，导致policy 多的时候GC会有一个很长的停顿时间。另外在 client 模式下，一台机器可能会起多个client，每一个 client 如果按照Ranger的原生模式，都会缓存一份 全量policy 到内存,这种情况下内存的损耗在多个client中是叠加的，我们的解决方案是做一个内存转网络的鉴权方案，统一把这种 client 模式的鉴权转成了网络的方式，如下图。

第二个改进是，我们对Ranger进行了垂直拆分。以前一个Ranger管理了多个集群，Ranger中一个 service 对应的就是一个集群，一个service其实就是对应一套元数据，一套元数据约等价于一套拥有唯一的Hive metastore的hadoop集群。我们对Ranger进行了垂直拆分，比如我们内部现在有九个集群，有三个Ranger来支撑，一个Ranger只管三个集群，这样单个Ranger的 policy 数量就下降了，GC 时间就得到了控制，Ranger admin 的性能也得到了保障。

这两个思路其实都不复杂，但是我们实践中还是遇到了不少的小问题。比如第一个问题是内存转网络的时候，如果服务端不可用，任务会不会受到影响，这涉及到高可用的问题。第二个问题是请求的性能问题，同机房的调用在我们的实践中基本上就是几毫秒，但是我们在对外商业化的过程中遇到了一个情况，存在有跨国的机房，甚至跨洲的机房，比如印度机房调用中国的机房，或者是欧洲的机房调用中国的机房。这种跨洲调用过程中，网络的延迟就会非常明显。我们之前是每一列都会去调用的，现在发现调用次数还是比较影响性能的，多个请求必须要压缩成一次调用，需要去优化，尤其是对于Spark，由于执行计划优化的不同周期中没有通信所以容易存在重复鉴权的情况。

3. 鉴权请求报文体优化

我们是把Ranger插件改造成了远程鉴权的模型，大部分的修改是基于以前ranger插件代码去修改的。改造过程中如果图省事，可能直接把以前内存的一些参数转成网络的参数，但这会带来很大的性能问题。因为Ranger鉴权内存参数很大，一个报文体可能就有几Mb，如果并发QPS有几百上千的时候，内存消耗非常快，实践中发现一个大宽表可能对应鉴权报文有4-5Mb。对这种报文体是必须要进行优化的，如果还是用 Ranger原生的内存鉴权报文就会太大，对性能损耗也会很大。

4. 冻结目录

2019 年，我们内部一些重要库的 HDFS 路径被误删。虽然误删之后，因为HDFS有一些垃圾回收站等机制，最终数据还是能恢复过来，但恢复的过程非常吃力，另外恢复过程中基本数据是不可用的，对业务的可用性影响很大。

除了组织上权限管理的问题以外，前面也提到过我们认为Ranger最大的一个缺陷就是它对目录的delete、 rename 这种删除、重命名操作的权限是含在其写的权限中的。这就可能存在一些隐患，因为删除权限级别本身应该是高于写的。我们的解决方案比较简单，就是把 HDFS 中的delete、 rename

这两个 action 单独抽离出来进行鉴权，不合在写的权限里面，最终我们的实现效果也是明显的。内部的核心库得到保护，路径不会再被误删。

5. 动态脱敏

为什么要增强ranger的动态脱敏？这里举了两个例子。第一个是如果一个数仓中有 100 列都是电话号码，开始是把电话号码全部配成了遮盖脱敏，突然需要电话号码把前三位露出来，只对后面的几位数字脱敏，这种情况下，原生的脱敏规则管理员可能就要改 100 次，成本很高。如果下次又要把电话号码的前四位露出来，又需要改 100 次。第二个是 Ranger 的policy没法作用在一个范围上（只能作用在具体的列上），导致无法配置对指定库不脱敏，如果需要配置管理员角色对一批库不脱敏，在原生的Ranger，至少 2. 1 版本的Ranger中是没法做到的。

我们认为第一个问题的主要原因是Ranger没有对规则的泛化能力必须指定一个具体的UDF函数。我们对Ranger 插件进行了优化，它不再依赖于一个具体的 UDF 函数，而是依赖一个泛化脱敏规则。比如我的一个脱敏规则就叫「电话号码脱敏」，它是一个泛化的规则，具体如何脱敏不确定。对电话号码的列，就配置为「电话号码脱敏」，具体对应的脱敏的UDF细节是可变的。脱敏的时候插件首先通过policy找到对应脱敏规则的ID，然后去服务端请求把 ID 换成具体的脱敏实例。最终吐给计算引擎的时候，再把它转成一个具体 UDF 的格式，这样就可以达到上文描述的效果。

第二个问题，我们认为Ranger的脱敏规则管理是有一定缺陷的。上图的左边可以看到Ranger原生的管理是通过优先级进行管理的，一列可有多个优先级不同的脱敏规则，排在最上面的优先级最高，比如角色a用规则1脱敏优先级最高，角色b用规则2脱敏优先级其次，当然这样是很灵活，但是管理成本也是非常高的，尤其是当用户在多个角色中的时候实际走到的是哪条规则情况就较为复杂。

我们的方案是把它转成了一种白名单的方式，比如配置a、b、c用户是白名单，不需要脱敏，对其余所有的用户都是用规则1脱敏这张表，这样我们只用去管理白名单的机制就可以了，用户也可以去独立申请白名单。

此时白名单只有两层优先级的概念了。白名单优先级是大于脱敏的，只要在白名单中，肯定不脱敏，退出白名单，肯定会脱敏。这样逻辑就简单化了，虽然可能略微丧失一定的灵活性例如不同角色有分级的脱敏规则，但是实践中发现该方案才是产品中实际能被用起来的一种方案，为了实现该目地需要修改对应的ranger 插件中脱敏policy匹配相关代码，加入范围的脱敏规则（例如配置在库.* 下的规则）匹配，以便实现在一个范围下配置白名单效果。

我们优化的最终效果就是脱敏的易用性得到了提升，同时扫描和脱敏是一体化的，从图右边可以看到，扫描任务可以自动地把扫出来的敏感类型推到脱敏里面去。扫描脱敏的一体化能够使得产品体验得到很好的提升，这也是单一ranger做不到的。

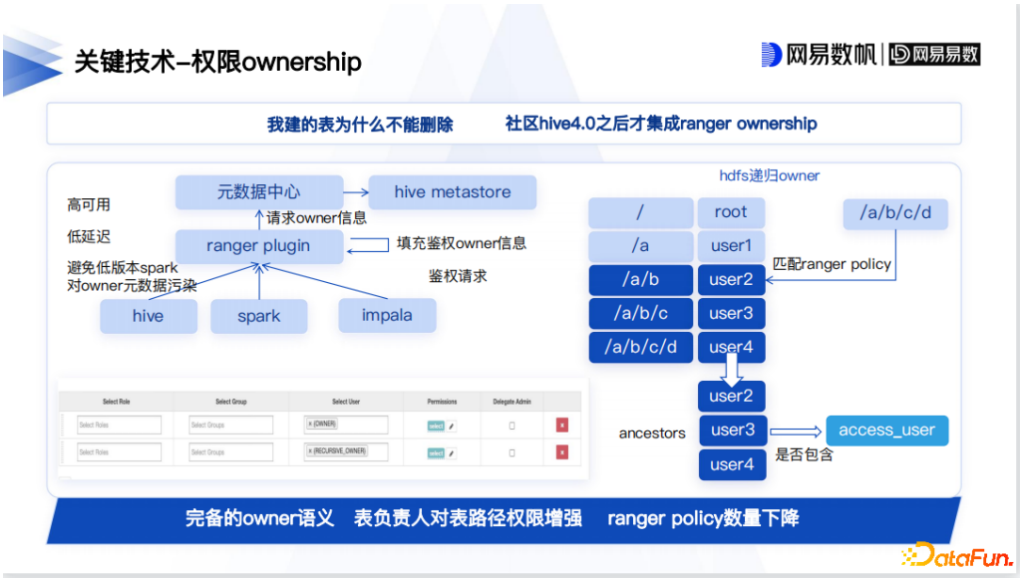
6. 审计和治理

这其实对于权限管理是一个老生常谈的问题了，做过权限的同学可能都会遇到一个问题就是用户突然找过来说：“我之前有权限，怎么突然就没有了，你帮我加一下”。这个时候可能研发只能帮用户把权限补上去了，但是也不知道什么原因。第二个问题是有多少权限是无效权限？权限授了，系统里面很多权限也没什么问题，但是到底有多少有用，很难判断。Ranger 也不好解决这两个问题，因为它没有一个全资源视角的审计，对 policy 变更的审计，只能是 policy 的视角，在 policy 删除重建等操作时就审计不到。同时它也没有一个 item 视角的层级，具体一个人到底是通过角色还是通过个人权限访问的资源是无法审计到的。

我们的解决方案就是针对这个问题，首先把权限模型和用户角色变更同步一份到数仓里去分析追踪。同时我们在Ranger审计插件的审计模块中做了一些改动,首先把 policy 鉴权过程中的命中信息发送到 Kafka 中去进行维度的补充，跟中台的维度进行了打通，这样我们审计可以提供所有的权限变化追踪，同时我们还补了一个角色命中的信息，通过什么角色去访问的信息也很重要。在治理中，通过维度的补充这个过程，我们就可以把它细化到具体的人或角色了，根据它给我们的原始数据的policy id，我们会把它拆成具体用户是通过哪一条item，哪一个具体的权限去访问的一个表，我们的细粒度就可以解决这个问题。

方案的最终效果明显，我们在内部定位发现，用户经常删表重建，重建后的表就没了访问权限，我们就清楚了这个问题的答案。同时我们发现内部有 70% 的权限是半年都没有用过的权限，这个比例是非常大的。但是无效权限或者冗余权限能不能直接回收，这是另外一个问题，可能冗余权限不一定就完全可以直接回收，我们还在实践过程中。

7. 权限的ownership



Ownership指的是表的负责人自动对表有所有权限，而不需要额外用 policy来授权。我们经常遇到的一个问题是用户反馈我建的表为什么不能删除，但Ranger原生的ownership机制并不完备，因为社区的 Hive 4.0 之后才集成了Ranger原生的 ownership，而Impala 应该是在较早的版本就集成了。所以 ownership 是一个比较棘手的问题。

我们的解决方案是基于元数据中心的 owner 的补充。我们会从鉴权的时候，去请求一个元数据中心，也就是内部的元数据系统，补充 owner 的信息，然后再走 Ranger原生的ownership鉴权逻辑。

为什么要走元数据中心，不能从底层组件把 owner 带过来？因为我们遇到过，低版本的Spark尤其是 2.3 版本以下，会对表 owner 的元数据进行污染。Hive meta store 里面的 owner 字段会被低版本的 Spark 污染。因此从上层来解决能够提供更好的兼容性。

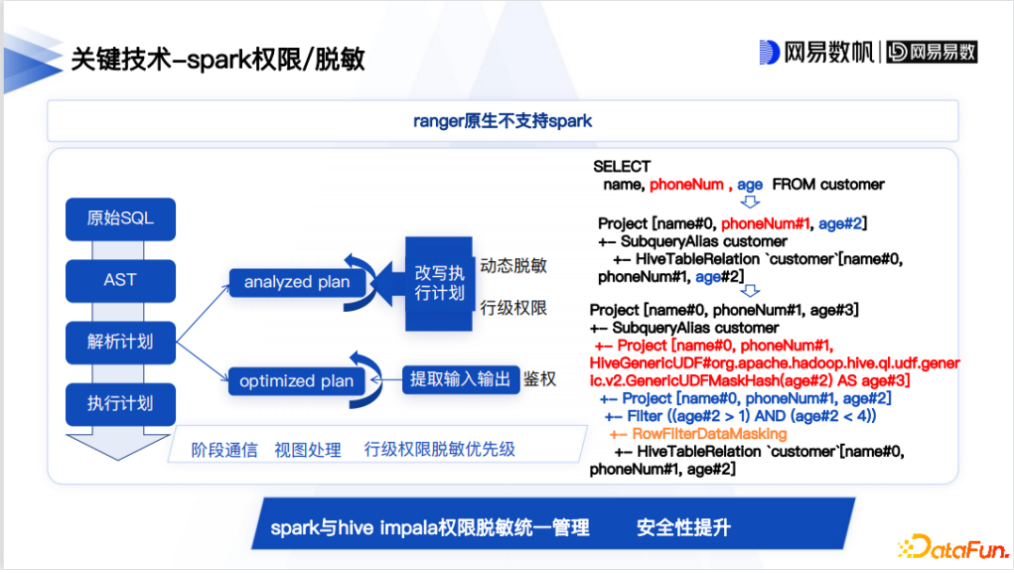
同时还有一个困难点是 HDFS 的 owner 问题。原生的 HDFS owner权限其实不包含递归属性，但是表的 owner 对表的路径应该是有递归的权限的，否则表下面的分区就不一定能访问，但是原生的 HDFS 的权限（acl）或者原生Ranger，都没法支持路径的递归的owner语义。

针对这个问题，我们对NN的鉴权插件进行了一个扩展，可以看到上图图左下面，上面 owner 是原生的owner，下面 recursive owner 是我们在Ranger上面新增的owner。recursive owner的含义就是这个路径的 owner对这个路径具有递归的访问权限。

实现的方案略微有点复杂。鉴权请求中的路径首先命中了一条 policy 之后，把所有的「请求路径」到「policy资源路径」之间的「祖先」owner 算出来，例如上图右图中的user2、user3、user4为「祖先」owner。然后通过「祖先」 owner 去跟当前请求的被鉴权用户匹配，如果「祖先」 owner 包含被鉴权用户则命中item中的recursive owner占位符，然后再看recursive owner 的item有什么权限就返回相应的结果。要对 Ranger 的插件进行一些修改，主要是NN的插件进行修改。通过这些手段我们就达成了一套很完备的owner语义。同时我们的表负责人对路径的权限也得到了增强，不存在表负责人对表路径只有非递归的权限了，目录下的子目录的权限都可以被继承。

最后一个点是Ranger的 policy 数量也会下降，因为以前的这种体系没有 owner 语义。其实我们每张表是有一条 policy 去表征其owner 的权限。在表的数量大的时候， policy 数量也很大。有了 owner 语义后，一条 policy 就可以表示出来所有的 owner 都有对应表的权限。

8. Spark权限/脱敏



Ranger原生是不支持 Spark 的，我们通过一些介入 Spark 的执行计划的过程来实现权限脱敏。首先是 Spark 的执行计划有四个阶段，在解析这个阶段又分为 analyzed执行计划和 optimized 执行计划，我们的动态脱敏和行级权限，其原理就是去修改 analyzed 执行计划，在该analyzed阶段中添加 filter 和脱敏函数，其实就是添加UDF。

鉴权是通过 optimized 阶段执行计划的一个输入输出提取去鉴权。但是在这种思路下有一些难点需要去处理，比如Spark每一个优化的阶段是反复执行的，不是一次优化就结束的。所以该思路存在阶段通信的问题，比如此阶段已经执行过脱敏了，下一个阶段再进 analyzed 的时候，就不能再脱敏了。权限其实也是一样的，只需要鉴权一次。这种阶段之间通信需要一个沟通或者通信的机制。其次就是视图的处理，Spark视图权限整体是比较复杂的，需要单独处理。

最后一个要点是这三者之间的优先级关系，行级权限和脱敏的优先级，到底是先脱敏之后再走行级权限，还是用原文去走行列权限再脱敏，这些都是实践中会遇到的一些问题。当然，目前问题我们也都已经解决了。可以看图右边的执行计划代码。其中红色的部分是一个脱敏的执行计划修改，蓝色的部分是一个行级权限的执行计划修改，橙色的部分是我们插入用来阶段之间沟通的虚拟节点，只要遇到这个节点，就认为执行计划已经被处理过了。可以看到我们是把行级权限放在最下面的，所以我们是要先过行级权限再过脱敏的，这样也是比较合理的。我们的最终效果是Spark 与 Hive以及Impala 权限脱敏实现统一管理，不再需要多次授权。这样，便利性和安全性都得到了提升。

9. Ranger社区优化

可能做Ranger的同学会比较感兴趣，如果深入去做产品化会发现Ranger 的小 bug 其实挺多的。这里我分享两个，也是相对是隐藏得比较深的、解决耗时比较久的两个bug。

第一个 bug 是Ranger2. 0 以后的版本中 RangerPolicyResponsitory 这个类用了finalize去释放内存。Java 的 finalize 关键字风险是很高的，因此用了以后，内存的释放很慢，本质上的原因是Responsitory与enricher的生命周期不同，Responsitory短于它所依赖的enricher的生命周期。Ranger开发者的解决方法是用 finalize 去释放。

但是这可能不是一个很好的方案，因为我们实践中发现，我们NN上线之后， fullGC 很频繁，基本上30-40分钟一次 fullGC，因为policy是全量内存缓存，尤其是Ranger policy 多的时候，内存可能有1-2G，释放太慢，每次又是全量更新，积累下来很快就 fullGC 了。我们知道 NN 是实时性很强的服务，一次 fullGC 对业务的影响是非常大的，所以我们线上不得不立即全量回滚，直到后面才解决这个问题。我们解决完之后，现在我们的NN的QPS 基本上4w+，峰值可能5w，基本上鉴权毫秒级还是非常稳定的。

第二个主要是右图代码所示的部分。

右图其实是两个并发不安全的问题，第一个问题，原生的Ranger可能对并发修改稍微有点薄弱,图中可以看到原生Ranger首先是拿到policy当前的一个版本号，在第二步里面，它就去把版本号+1，再去设置成新的版本号。1和2之间是没有任何加锁和互斥的，这肯定是不安全的，两个请求并发进1和2之间，会导致线程不安全，版本号肯定是会错位的。

第二个问题，如图所示，我们可以看到Ranger的缓存更新操作，在更新缓存的时候，首先是设置到当前缓存的版本号，再去设置缓存的内容，如果在步骤1和2之间有读请求进来会比较麻烦，这可能就会发生一次脏读，这也是不安全的，因为版本号已经上去了，但是policy内容还没更新好，读出来的就是一个新的版本号，但是 policy 内容是旧的，如果后面继续增量更新，就会基于一个错误视图去更新，就会永远漏掉一些policy。

10. 商业化

最后，分享网易数帆遇到的一个特殊的场景。

网易数帆也是一个 ToB 商业化的平台，我们不光要解决网易内部的安全问题，也致力于解决企业级服务的大数据安全问题。在大范围内铺开过程中，我们遇到了两个问题，第一个是依赖关系有点复杂，可以看到图中橘色的部分是我们安全团队负责，但是蓝色的部分，这些计算引擎又是其他底层团队负责的。我们两个团队之间的依赖变成了犬牙交错的状态。

这种犬牙交错的状态会带来一个问题，就是我们对外多环境部署的时候，配置依赖比较复杂，同时我们版本升级变更频繁的时候，变更顺序也比较复杂，容易出现升级出问题或者配置丢失的情况。因此在版本上必须要做好协同管理。同时我们安全中心也做好了兼容性保证，比如底层引擎没升级，安全中心即橘色部分会去兼容一些场景。

图右边是想说明这样一个问题，我们这套解决方案由于商业化场景客户的多种多样，我们底层 Hadoop 不一定是我们自己的Hadoop，有可能是用户自己提供的一套Hadoop，也有可能是CDH、CDP，什么都有可能。在这种情况下，我们的功能也是可以实现的，因为我们的解决方案是一套相对通用的数据安全解决方案。当然如果在用户使用自己Hadoop 的情况下，也会损失一部分能力，但是我们安全的核心能力还是可以保证的。

这两个点其实就要求我们的安全中心要做一个抽象，需要把不变的东西抽出来，从而随机应变，这对设计会有一定的要求。我们现在在做的30来个环境整体上部署变更都比较敏捷。其中很大比例也采用了对接模式，即用户使用自己的Hadoop，我们也都能顺利承担。

04 成果

最后简单分享一下我们的成果。

内部主要应用在网易云音乐、网易严选等，更多的是外部商业化。我们方案的稳定性和产品的易用性都已得到了很好的验证。

分享嘉宾

吴俣，网易数帆 资深开发工程师。华中科技大学信息安全专业毕业，目前在网易数帆负责大数据安全中心开发工作。

安全运营 (https://www.secrss.com/articles?tag=安全运营)

相关资讯

2023年API安全六大威胁和五个最佳实践 (https://www.secrss.com/articles/58066)

安全运营 (https://www.secrss.com/articles?tag=安全运营) · GoUpSec (https://www.secrss.com/articles?author=GoUpSec) · 2023-08-23

贴膜场景下的抗打印数字水印 (https://www.secrss.com/articles/58056)

安全运营 (https://www.secrss.com/articles?tag=安全运营) · 隐者联盟 (https://www.secrss.com/articles?author=隐者联盟) · 2023-08-23

网络安全风险管理的10大关键要素 (https://www.secrss.com/articles/58029)

安全运营 (https://www.secrss.com/articles?tag=安全运营) · 安全牛 (https://www.secrss.com/articles?author=安全牛) · 2023-08-22

评论（0）

登录后才能发表评论，请先 登录 / 注册 (https://www.secrss.com/login)